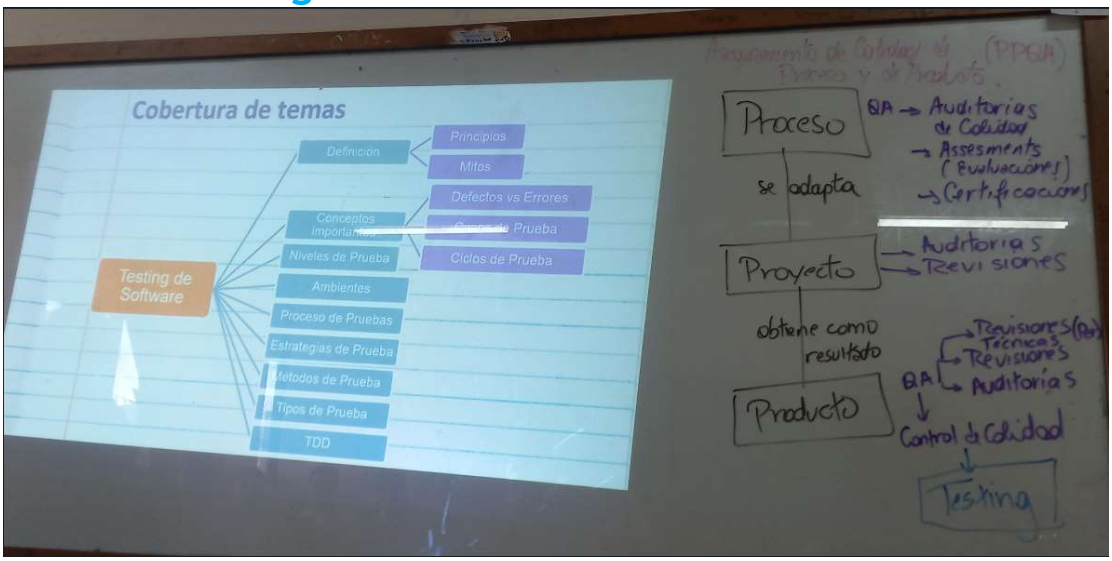
Testing en ambientes Ágiles.

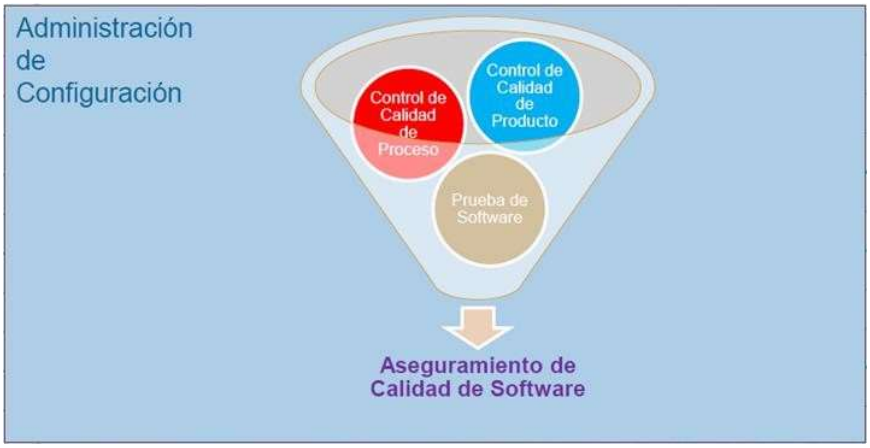


Contexto

El testing de software, es una de las tareas a realizar en el ámbito del aseguramiento de calidad de software (PPQA), en conjunto con el control de calidad del proceso y del producto. La calidad del software se obtiene y se logra a lo largo de todo el desarrollo de este (detectar errores en etapas tempranas, es siempre menos costoso, que detectarlos después), por lo que una buena administración de configuración (SCM) es el puntapié inicial para permitir la realización de tareas de aseguramiento de calidad.

Hay que recordar que SCM permite determinar la configuración de ítems, trazabilidad, control de cambios, auditorías del producto y reportes de estado. QA(Quality Assurance) agrega a esto, el control de calidad del proceso. Existen diversos estándares (ISO, CMMI, etc.) que permiten acreditar la calidad del proceso, debido a que la calidad del producto no es algo estandarizable por la variación de requerimientos de un caso a otro.

Dentro del aseguramiento de la calidad del software, existen actividades destinadas al control de calidad del proceso y del producto. Se realiza un control de calidad sobre el proceso de desarrollo, bajo el argumento de que el proceso de desarrollo posee un gran impacto en la calidad del producto. Es decir, en teoría la calidad del producto depende de la calidad del proceso que se está utilizando.



Luego de desarrollar el software, las actividades de testing revelan la calidad del Software en sí, es decir, si el producto satisface con los requerimientos definidos por el cliente en etapas anteriores. Cabe destacar que la actividad de testing no asegura la calidad por sí solo, sino que debe estar acompañada de las demás actividades.

QA(Quality Assurance) != Testing

Definición de Testing

Proceso mediante el cual se somete a un software o un componente de software a condiciones específicas con el fin de determinar y demostrar si el mismo es válido o no en función de los requerimientos especificados.

El testing es una actividad destructiva cuyo objetivo es encontrar defectos, cuya presencia es asumida de antemano en el código.

Testing de software es un proceso, o una serie de procesos, diseñados para asegurar que el código hace lo que fue diseñado para hacer, o visto de otra manera, que no haga nada que no esté intencionado. Se dice que es el proceso de ejecutar un programa con la intención de encontrar fallas. El software debería ser predecible y consistente, sin presentarle sorpresas a los usuarios.

El Testing es parte del QA (Quality Assurance) y forma parte del control de calidad. Se hace cuando el producto ya está construido (o lo que se desea testear), en cambio el QA se aplica en todo el ciclo de vida (las buenas prácticas en la implementación son parte de QA).

Las actividades de testing se consideran exitosas si se encuentran defectos en la ejecución de los casos de prueba. Por lo que, un software con alta calidad lograda a lo largo de todo el desarrollo (implementación de QA), dificulta encontrar defectos y puede dirigir a un testing no exitoso. Por otro lado, si el software tiene un bajo nivel de calidad, el costo de retrabajo es alto, ya que van a existir muchos idas y vueltas entre las actividades de testing y las de desarrollo, para la corrección de errores.

El testing es una actividad costosa, por lo que es necesario lograr un nivel de cobertura óptima teniendo en cuenta el costo-beneficio. Nunca se podrá cubrir el 100% de las pruebas, por cuestiones lógicas de tiempo y costos. Esta cobertura se comienza a definir desde el momento en los cuales se establecen los criterios de aceptación.

Principios del Testing

- El Testing es una actividad destructiva que encuentra defectos cuya presencia se asume.
- Se testea con una actitud negativa tratando de demostrar que algo es incorrecto.
- El Testing exitoso es aquel que encuentra defectos.
- El costo del Testing está entre un 30% y un 50% del valor del producto.
- El Testing pone en evidencia defectos, pero no agrega calidad ni garantiza que el producto no tiene errores.
- Un desarrollo exitoso puede llevar a un Testing no exitoso.
- El Testing es necesario siempre.
- Se parte de la suposición de que siempre se tendrá defectos por encontrar, ya que es una característica inherente de los productos desarrollados por equipos de personas.
- Puede empezar antes de la codificación porque necesita de los requerimientos para armar los casos de prueba.
- El testing NO asegura que se tenga un producto de calidad (ni la agrega), ni que el proceso por el que se desarrolló sea de calidad.
- Agrupamiento de defectos: los defectos en el software suelen agruparse en un conjunto limitado de módulos o áreas (regla de Pareto, 80% de los defectos se agrupan en el 20% de la funcionalidad). Bajo este principio, las pruebas deben priorizarse y enfocar el esfuerzo en ese conjunto acotado de funcionalidades.

Principio	Explicación
Una parte necesaria de un caso de prueba es definir el resultado esperado.	Si el resultado esperado de un caso de prueba no ha sido predefinido, lo más probable es que un resultado erróneo se interpretará como un resultado correcto, debido a el fenómeno de "el ojo viendo lo que quiere ver", a pesar de la definición destructiva adecuada de prueba, hay todavía un deseo subconsciente de ver el resultado correcto.
Un programador debe evitar testear su propio programa.	Los propios desarrolladores “no quieren” encontrar sus propios defectos, por lo que el carácter de pruebas destructivas se deja de lado. Además, puede que el programa contenga errores por malentendidos del programador sobre el dominio, y si él mismo testea, no se van a detectar estos defectos.

Una empresa de desarrollo no debe testear sus propios programas.	Misma explicación que el principio 2 orientado a empresas, agregando que si las mismas empresas testean sus programas, es posible que destinen menos recursos y eviten encontrar defectos para cumplir con el calendario y los costos establecidos.
Cualquier proceso de prueba debe incluir una inspección minuciosa de los resultados de cada prueba.	Se deben realizar inspecciones minuciosas para detectar la totalidad de los defectos encontrados tras la ejecución de una prueba, ya que pueden surgir más de un defecto (y esto es lo que se busca) por cada caso de prueba.
Los casos de prueba deben escribirse para condiciones de entrada que no son válidas e inesperadas, así como para las que son válidas y esperadas.	Muchos errores que se descubren repentinamente en el desarrollo de software aparecen cuando se usa de alguna manera nueva o inesperada, y no responde cómo debería (catcheando el error)
Examinar un programa para ver si no hace lo que se supone que debe hacer es sólo la mitad de la batalla; la otra mitad es ver si el programa hace lo que no se supone que debe hacer	Los programas deben ser examinados para detectar defectos secundarios no deseados.
Evite los casos de prueba desechables, a menos que el programa sea realmente un programa de descarte.	Los casos de pruebas utilizados en el testing deben ser reproducibles, es decir, no se deben realizar casos de prueba sobre la marcha (ad-hoc) ya que es imposible reportar un defecto sin tener las condiciones y los pasos en los cuáles el defecto surgió. Además, realizar testing es muy costoso, por lo cuales los casos de prueba deben ser reutilizados para volver a testear escenarios luego de la corrección de errores.
No planee un esfuerzo de prueba bajo la suposición tácita de que no se encontrarán errores.	Es un error pensar de esta manera al momento de realizar software. Se debe tener en claro la definición de testing, en la cual se define que el objetivo de esta actividad es encontrar errores, y se presume de antemano su existencia.
La probabilidad de que existan más errores en una parte de un programa es proporcional al número de errores ya encontrados en esa parte.	El concepto es útil porque nos da una idea de en qué sección del programa hacer foco o asignar más recursos, si una sección particular de un programa parece ser mucho más propenso a errores que otras secciones, es recomendable realizar pruebas adicionales y es probable que encontremos más errores.
Las pruebas son extremadamente creativas e intelectualmente desafiantes.	Aunque existen métodos y estrategias para abarcar un mejor nivel de cobertura testing en los casos de pruebas, siempre es necesario un poco de creatividad del diseñador de estos.

Mitos del Testing

- "El testing es el proceso para demostrar que los errores no están presentes".
- "El propósito del testing es demostrar que un programa realiza sus funciones previstas de forma correcta"
- "El testing es el proceso que demuestra que un programa hace lo que se supone que debe hacer''

Estas definiciones o afirmaciones sobre el testing son incorrectas, ya que el objetivo de testear un programa es agregar valor al producto revelando su calidad y brindando confianza en el software, de forma más concreta, encontrar y remover defectos en el código de este. Entonces, no se prueba un sistema para mostrar que funciona, sino que se comienza con la suposición de que el software contiene defectos, y se realiza el testing para encontrar la mayor cantidad de ellos posible.

Por otro lado, un programa puede hacer lo que se supone que debe hacer, y aun así, contener defectos. Es decir, un error está claramente presente si un programa no hace lo que se supone que debe hacer, pero los errores también están presentes si un programa hace lo que no se supone que debe hacer.

¿Cuánto Testing es suficiente?

El **testing exhaustivo es imposible** por la cantidad de tiempo que requiere. El momento en que se deja de hacer testing depende del nivel de riesgo o costo asociado al proyecto. Los riesgos permiten definir prioridades de que se debe testear primero y con qué esfuerzo.

El criterio de aceptación se utiliza normalmente para decidir si una determinada fase de testing ha sido completada. Este puede ser definido en términos de:

- Costos.
- % de tests corridos sin fallas.
- Inexistencia de defectos de una determinada severidad.
- Pasa exitosamente el conjunto de pruebas diseñado y la cobertura estructural.
- Good Enough: Cierta cantidad de fallas no críticas es aceptable.
- Defectos detectados es similar a la cantidad de defectos estimados.

El criterio de aceptación sirve para definir y negociar en ágil con el Product Owner y en tradicional con el Líder del Proyecto a cuántos defectos son aceptables para terminar.

Conceptos importantes

"Defecto" versus "Error"

La diferencia entre ambos conceptos es el momento en el cual se detectan y solucionan los errores o defectos. Un error es detectado y corregido en una misma etapa, y un defecto es un error que se traslada de una etapa a otra etapa posterior en la cual se introdujo. El testing encuentra defectos, ya que son errores que surgieron en etapas anteriores, pero se detectan en la etapa de prueba.

Hay que aclarar que ambos conceptos pueden o no generar fallas en el sistema, es decir, un mal funcionamiento de este.
Por ejemplo, un error en los colores de una interfaz no es una falla, ya que no conllevan a un mal funcionamiento del sistema.

Defecto

Un defecto posee dos características principales, que permiten catalogarlos en la etapa de testing:

- Severidad: define la gravedad del defecto, y es determinada por la persona que realiza el testing, por lo que esta característica es de carácter técnico. El valor de la severidad se asigna dependiendo de la siguiente escala:

    - Bloqueante:
      El defecto no permite continuar con la ejecución del sistema.
    - Crítico :
      El sistema funciona, pero la funcionalidad que se está testeando tiene un defecto crítico.
    - Mayor
      La funcionalidad que se está testeando funciona, pero no de forma correcta.
    - Menor
      La funcionalidad se ejecuta correctamente, pero con advertencias erróneas o errores de baja importancia.
    - Cosmética:
      formato de fechas, formato de números, distribución de componentes en una GUI, temas de presentación de interfaces, etc.

- Prioridad: la prioridad define el impacto del defecto en la funcionalidad para el negocio, y permite ordenar la atención de los defectos según las necesidades del cliente. Una escala posible para la prioridad puede ser:
    - Urgencia
    - Alta
    - Media
    - Baja

Estas características son independientes entre sí, ya que varían dependiendo del tipo de negocio y de sistema que se está desarrollando. Por ejemplo, un defecto cosmético puede tener baja prioridad para un sistema de inventarios, pero puede tener una alta prioridad si se trata de un sistema de una empresa de marketing, en donde el aspecto y la presentación es algo muy importante.

Casos de prueba

Son una secuencia de pasos que hay que seguir para obtener un resultado esperado, y se tienen que dar ciertas condiciones previas que fijan una situación para que se pueda ejecutar la prueba.

Es el artefacto más importante del Testing. Este se hace una vez, pero sirve para realizar infinitas pruebas, debiendo obtener en todas el mismo resultado esperado.

Se trata de minimizar la cantidad de casos de prueba maximizando la cantidad de defectos encontrados.

El objetivo de los casos de prueba es descubrir defectos.

Los casos de prueba salen de los requerimientos, permitiendo hacer tanto la verificación como la validación.

Está compuesto por: un objetivo (lo que se desea controlar), condiciones de prueba (Datos de entrada y de entorno que deben estar presente para llevarse a cabo la prueba) y un resultado esperado.

Ciclos de prueba

Es la ejecución de un conjunto de casos de prueba en una versión determinada del producto. Generalmente se tienen 2 ciclos, y el primero es conocido como ciclo 0. El ciclo 0 siempre es manual, es donde se configura todo y a partir del ciclo 1 ya se pueden automatizar las pruebas.

En caso de detectarse defectos, el producto vuelve a desarrollo para su corrección.

Si existe una cantidad alta de ciclos de prueba, es probable que QA no sea bueno, por lo que deberían revisarse ciertos aspectos acerca de la calidad del proceso y producto, de manera de evitar tener grandes cantidades de ciclos, que se traducen en mucho retrabajo, lo cual a su vez conduce a incrementar los costos de desarrollo.

Ciclo de pruebas con regresión

Este concepto plantea la necesidad de ejecutar exhaustivamente todos los casos de prueba en cada ciclo de prueba, como si fuesen el ciclo 0. Este planteo es el ideal, pero el menos utilizado, por cuestiones de practicidad y tiempo.

El otro enfoque, consiste en aplicar una prueba exhaustiva de todos los casos de prueba en el ciclo 0, pero a partir del

ciclo 1 (si el incremento o producto vuelve nuevamente a testing) ejecutar sólo los casos de prueba asociados a defectos encontrados previamente, y no los casos de prueba que hayan pasado sin encontrar defectos. Esto permite agilizar el proceso de pruebas, pero estadísticamente es muy probable que al arreglar un defecto reportado por testing, en el desarrollo de su corrección se introduzcan nuevos errores en otras partes del producto. Por lo que, si no se prueban todos los casos, a pesar de que previamente no se hayan detectado defectos, pueden encontrarse nuevos, debido al proceso descrito anteriormente.

Una solución a esta encrucijada es no aplicar regresión (ejecutar sólo los casos de prueba con defectos asociados), y cuando finalmente estén todos los defectos "corregidos", hacer una ejecución de todos los casos de prueba (como si se aplicara regresión), para aumentar la confianza de que no se han introducido nuevos errores en las correcciones.
La automatización del testing permite hacer tantos ciclos de prueba con regresión como se deseen. La desventaja es el proceso de automatizar la ejecución de los casos de prueba, pero se puede ver ese esfuerzo de automatización como el esfuerzo propio del ciclo 0 manual, y luego esa automatización permite ejecutar los casos de prueba N veces. En empresas donde el core de negocio no es hacer software, conviene asumir ese esfuerzo y automatizar las pruebas.

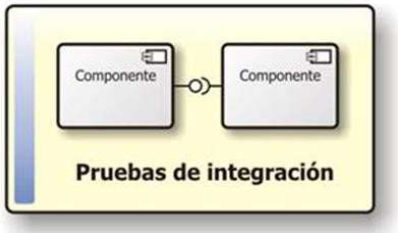
Claramente, la mejor estrategia es aplicar regresión, porque no existe un punto medio al ser un método de caja negra, ya que no se puede saber con certeza lo que va a afectar al corregir un defecto y en dónde impactará esa corrección. Sin embargo, en la práctica se utiliza sin regresión, debido a los costos y tiempos que implica ese proceso.

Niveles de prueba

Los niveles de prueba determinan el foco o la granularidad de la prueba que se está por ejecutar. En general, primero se comienzan probando componentes pequeños, y luego se integran en pruebas de mayor granularidad. Esto es al revés de cómo se desarrollan los componentes.

- Pruebas unitarias:
  Las hace el desarrollador y están incluidas en el DoD. Son pruebas que se realizan sobre un componente, de manera independiente a los otros. Se hacen teniendo acceso al código fuente, y se pueden usar herramientas para realizarlas (automatización, depuración). Se suelen reparar los errores apenas se encuentran, sin registrarlos formalmente.

- Pruebas de integración:
  El propósito de la ejecución de estas pruebas es verificar el funcionamiento de dos o más componentes juntos que se relacionen entre ellos, es decir, clases en las cuales existe una interacción a modo de peticiones, a través de sus interfaces (boundaries). El resultado de estas pruebas es el build (el CI o continuous integration), el cual luego se envía al equipo de testing y es lo que se prueba en siguientes etapas.



  Se suele llevar a cabo una prueba de integración incremental, desde lo más general (que abarca más componentes) a lo más específico (top-down) o desde lo más específico a lo más general (bottom-up).

- Pruebas de sistema o de versión: En estas pruebas se realizan testeos sobre la versión de un incremento de producto o de un producto (dependiendo si se usa Agile o métodos tradicionales), y existen razones psicológicas que demuestran que deben realizarlas personas ajenas al desarrollo de los programas (obligatoriamente). Estas pruebas se llevan adelante siguiendo un proceso sistemático y metodológico, que permite encontrar defectos que puedan ser reproducibles y reportar de qué manera ocurre el defecto detectado, es decir, cual es el camino de pasos que hay que seguir para encontrar el error.

Estas pruebas están basadas en casos de prueba. Se trata de emular de la mejor manera posible un entorno de trabajo idéntico al entrono real en el que se usara el SW. Se llevan a cabo pruebas del funcionamiento real y cotidiano del producto. Abarca requerimientos funcionales y no funcionales.

- Pruebas de aceptación de usuario: se realizan en el despliegue y debería realizarlas el usuario para verificar que se cumple con lo que el mismo requirió. Busca que el cliente/usuario se familiarice con el producto, generando comodidad y confianza sobre el mismo (su objetivo principal no es encontrar fallas). También busca reproducir un entorno de producción. Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (pruebas alfa), como la prueba en ambientes de trabajo reales (pruebas beta). En la teoría, los usuarios arman sus pruebas de usuario. En la práctica, las arma Testing.

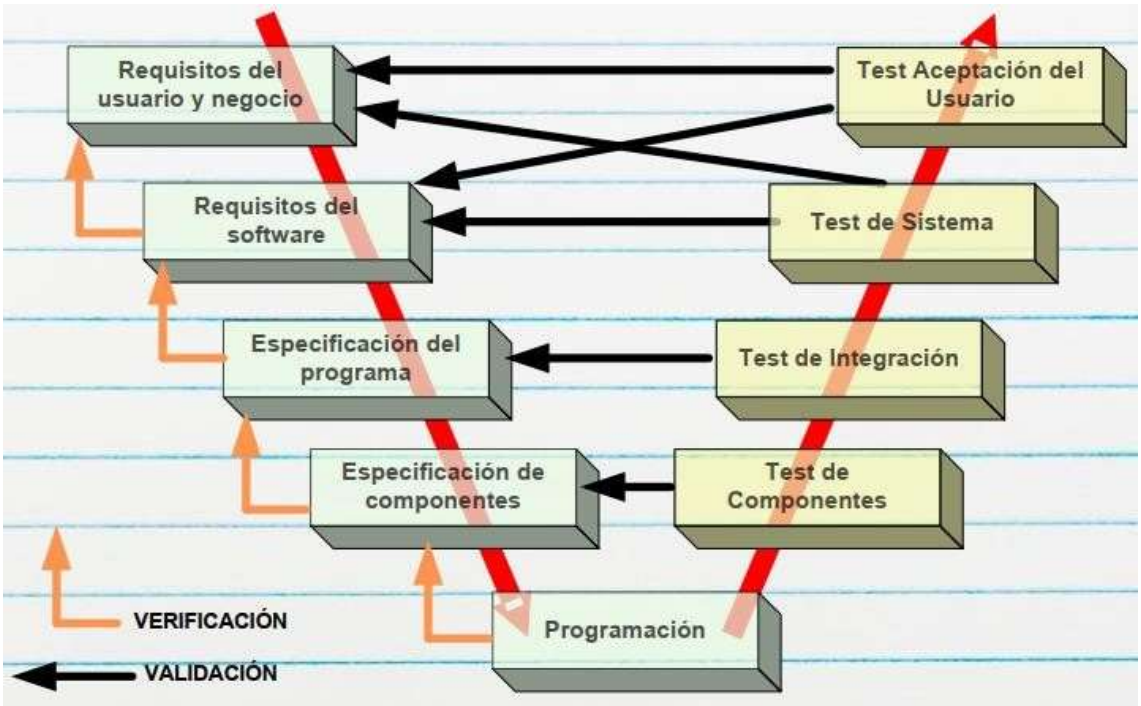
En la teoría, además del usuario, deben estar presentes el gerente de testing, el líder del proyecto y empleados con roles funcionales.
En Agile, además del usuario, deben estar todos los miembros del equipo (Sprint Review).

Modelo en V

Este modelo se utiliza para poder determinar cuándo se está en condiciones de realizar determinadas actividades, en función de la etapa de desarrollo en la que se encuentre el software.

Acá se puede notar cómo las pruebas se realizan en orden de granularidad inverso al proceso de desarrollo del software. Es decir, se desarrolla desde componentes de granularidad gruesa, como son los requerimientos abstraídos de detalles de implementación, hacia componentes de menor granularidad, como son clases o módulos, dependiendo del paradigma. En cambio, para las pruebas, la granularidad se da en sentido inverso.

- Testing de Componentes → Testing unitario.
- Especificación del programa → diseño/arquitectura.



Ambientes de Testing

Son todos los recursos, tanto hardware como software, que se requieren para poder trabajar con el producto. Existen distintos ambientes en el desarrollo de software, los cuales poseen distintas necesidades, según la etapa de desarrollo en la que se encuentre el software.

Desarrollo

Implica hardware y software necesarios (librerías, extensiones, IDEs, compiladores, etc.) para poder desarrollar y desplegar el producto y utilizarlo. En este ambiente se realizan las pruebas unitarias (y también las de integración generalmente).

Pruebas

Es el ambiente que utilizan los testers para llevar a cabo las pruebas, y los desarrolladores no deben tener acceso. En este se realizan las pruebas del sistema. Normalmente acá se realizan las pruebas de sistema.

Preproducción

Debería tener las mismas características que el ambiente productivo para poder comprobar que el producto funcionará una vez desplegado en producción de manera correcta. En este ambiente se realizan las pruebas de aceptación.

El problema de este entorno es que muchas veces resulta difícil (o imposible en algunos casos), debido a que es muy costoso replicar principalmente las configuraciones de hardware. Por ejemplo, sería prácticamente imposible replicar la configuración de servidores del motor de búsqueda de Google, para realizar pruebas de aceptación.

Producción

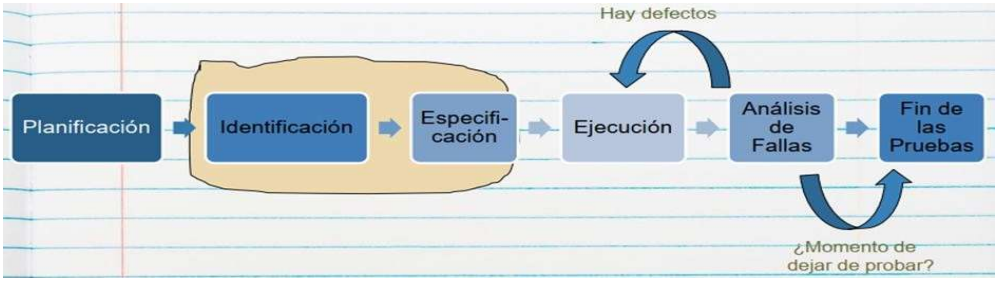
Este entorno, es la configuración de software y hardware que tienen los usuarios finales del software, y que utilizan para el desempeño de sus actividades laborales. Obviamente en este entorno no se realizan pruebas (probar en este ambiente tiene grandes consecuencias), ya que en teoría la versión del producto cumple con los criterios del Definition of Done.

Proceso de pruebas

El Testing es un proceso definido (que se lleva a cabo durante todo el ciclo de vida del producto), por lo que está compuesto de etapas detalladas.

Testing Ad Hoc: se realiza testing sin ningún proceso definido, perdiendo la trazabilidad ya que no se registra el paso a paso de lo que se probó, sino que se prueba libremente sin llevar ningún registro.

Generalmente el proceso de pruebas es estándar (puede tener algunas variaciones, pero sigue una estructura general).



- **Planificación**: Determina como se incluirá el Testing en el plan del proyecto, y como será el Test Plan (que recursos se usará, que riesgos se tendrá, cuál será el criterio de aceptación, que entornos se emularan, quien realizara cada Testing, cuando, etc.). El resultado de la planificación es el Plan de Pruebas, que debe contener:
    - Riesgos y objetivos del Testing.
    - Estrategia de Testing.
    - Recursos.
    - Criterio de Aceptación.

- **Diseño (Identificación y especificación de casos de prueba)**: Revisando las bases de la planificación del Testing, se identifican los datos necesarios, diseñan y priorizan los CP que se llevaran a cabo (se define que entorno se usara, como se llevaran a cabo, por quien, cuando, etc.). Analiza si los requerimientos son testeables o no. Se define si se usa regresión o no.

- **Ejecución**: Cuando se ejecutan los CP, se registran y se comparan los resultados generando un reporte de defectos (con defectos encontrados, condiciones, entornos). Se trata de automatizar lo más que se pueda respecto de estas ejecuciones (ahorrando tiempo/costo).
  Incluye: Creación de los datos necesarios para la prueba, automatización de todo lo que sea necesario, implementar y verificar el ambiente, ejecutar los casos de prueba, registrar los resultados de la ejecución y comparar los resultados reales con los esperados.

- **Análisis de fallas (Evaluación y Reporte)**: Se hace un seguimiento de la corrección de los defectos encontrados hasta que se cierren todos los CP, es decir que se hayan solucionado todos. Se evalúa el criterio de aceptación, se reporta el resultado de las pruebas a los interesados, se verifica los entregables y que los defectos se hayan corregido y se evalúa cómo resultaron las actividades de testing y se analizan las lecciones aprendidas.

  Acá se confecciona el informe de reportes.

- **Fin de las pruebas:** En las empresas más maduras se deja de probar recién cuando no hay defectos bloqueantes, críticos, mayores y los defectos menores y cosméticos son muy pocos, los que se ponen en la nota de Release. Se confecciona el informe final (opcional).

Artefactos de Testing

Plan de pruebas

Este plan, es análogo al plan de proyecto que se elabora en el enfoque tradicional (no forma parte de ese plan). En este plan, se definen:

- Datos necesarios para las pruebas.
- Recursos destinados a las pruebas.
- Herramientas a utilizar.
- Si se aplicará regresión (o no).
- Etc.

Para la confección de este plan no es necesario el código, sino que sólo se necesitan los requerimientos, en el formato que sea que se encuentren (ERS o Casos de Uso en métodos tradicionales, User Stories en Agile, etc.)

Casos de prueba

- Es el artefacto más importante del testing, y consiste en una secuencia de pasos a seguir, las cuales describen un conjunto de acciones a realizar para lograr un resultado concreto y esperado. Para esto se deben explicitar las condiciones de inicio (las cuales fijan una situación particular), y los datos utilizados en ese escenario.
- Se confeccionan con el objetivo de descubrir la mayor cantidad de defectos y minimizar el esfuerzo y tiempo para elaborarlos y ejecutarlos.
- Mientras no cambien los requerimientos, estos casos pueden ser utilizados las veces que sea necesario.
- La característica más importante de los casos de prueba es ser REPRODUCIBLES. Si se encuentra un defecto y no es reproducible a través de un caso de prueba, no es un defecto

Los casos de prueba se derivan de diferentes fuentes:

- Documentos del cliente;
- Información relevada;
- Requerimientos;
- Especificaciones de programación;
- Código.

Reporte de incidentes o defectos

Luego de la ejecución de los casos de prueba, se elabora un reporte, indicando cuáles fueron los casos de prueba que han pasado, y aquellos en los que se ha encontrado defectos, indicando cuáles fueron esos defectos encontrados y cómo reproducirlos para su detección y corrección.

Este artefacto permite a los desarrolladores corregir esos defectos, para enviarlos nuevamente a testing.

Informe final

suele contener métricas e información final del incremento del producto, se reporta cuántos ciclos y la cantidad de errores por ciclo, métodos, tipos de prueba utilizados, etc. Este es el único artefacto que es negociable en algunas organizaciones informales, se acuerda en la contratación del trabajo.

Tiene utilidades estadísticas.

El Testing y el Ciclo de Vida

Si desarrollamos con ciclo de vida en cascada (Ciclo de Vida Secuencial) las pruebas se hacen al final ya que es el momento en el que nos entregan el producto. En cambio, sí desarrollamos con ágil (Ciclo de Vida Iterativo/Incremental), hacemos testing por iteración. El problema es que se pueden incluir errores por la integración de los incrementos.

Estrategias de prueba

Se utilizan para pasar por la mayor cantidad de funcionalidades con la menor cantidad de casos pruebas confeccionados, debido a que el tiempo y el presupuesto para diseñarlos y ejecutarlos es limitado.

Hay dos grandes enfoques: Caja Negra y Caja Blanca. Cada uno tiene fortalezas y debilidades particulares: un método puede ser bueno para algunas cosas, y no para otras cosas. El mejor método es no usar un único método, sino usar una variedad de técnicas ayudará a un testing efectivo.

Caja Negra

En esta estrategia no se dispone de la estructura interna de la implementación, sino que se analizan las funcionalidades como una caja negra, en términos de entradas y salidas de esa “caja”.

El proceso consiste en ingresar determinados datos a esa funcionalidad vista como caja negra, y luego comparar los resultados obtenidos con los resultados esperados. Para ello, se realiza un testing exhaustivo de entrada, probando cada posible entrada conducida como caso de prueba, no necesariamente las válidas. En transacciones no sólo se debe considerar todos los datos posibles, sino todas las secuencias posibles de transacciones previas.

Se clasifican en basados en especificaciones y basados en experiencia:

- Métodos basados en especificaciones:
  Son aquellos que se ejecutan utilizando la documentación de especificaciones realizadas del producto.
  - *Partición de equivalencias*:
    Proceso sistemático que consiste en identificar clases de equivalencia, que definen subconjuntos de datos que producen un resultado equivalente.
  - *Análisis de los valores limites*:
    Variante del anterior. Utilizar los limites o valores de borde de las clases de equivalencia para la definición de los casos de prueba.

- Métodos basados en experiencia:
  La experiencia y los conocimientos del tester son fundamentales para determinar las entradas del sistema y analizar los resultados.
  - *Adivinanza de defectos*:
    Enfoque basado en la intuición y experiencia para identificar pruebas que probablemente expongan defectos del software, elaborando una lista de defectos posibles o situaciones propensas a error y realizando pruebas a partir de esa lista.

- Testing exploratorio:
  El tester mientras va probando el software, va aprendiendo a manejar el sistema y junto con su experiencia y creatividad, genera nuevas pruebas a ejecutar.

Partición de equivalencias

Analiza las condiciones externas (entradas y salidas) involucradas en una funcionalidad determinada. A esas condiciones externas las divide en clases de equivalencia, las cuales son subconjuntos de valores posibles, que arrojan resultados equivalentes en la funcionalidad que se quiere probar.

- Ejemplos de entradas: campos de texto, fechas, coordenadas de un mapa, archivos, señales de sensores, etc.
- Ejemplos de salidas: mensaje en pantalla, advertencias, listados, tablas, emisión de señal, etc.

Procedimiento

1. Identificar condiciones externas de entrada y salida.
   a. Si hay que ingresar dos datos que son iguales, pero con distintos valores, separarlo en condiciones externas diferentes.
   b. Por ejemplo:
      2 calles, una condición externa es calle1 y la otra calle2.
2. Identificar subconjuntos de valores posibles (válidos y no válidos) de las condiciones externas identificadas, que produzcan resultados equivalentes en la ejecución de la funcionalidad.
   a. Los subconjuntos no válidos tienen asociados mensajes de error en la condición externa de salida "Mensajes de Error" o "Mensajes de Advertencia" (no es necesario poner todos los mensajes, con algunos genéricos es suficiente).
   b. Para no olvidarse de algún subconjunto → la unión de todos los subconjuntos debe ser igual al universo de la condición externa (teoría de conjuntos).
3. Armar los casos de prueba tomando de cada condición externa, un sólo valor particular (especificar) de cada subconjunto de valores posibles.

Consideraciones

- Prioridad
  - Alta → caminos felices o errores críticos en el dominio de negocio.
  - Baja → casos de prueba relacionados a validaciones (valores no ingresados, con mal formato, etc.).
- Nombre caso de prueba:
  Describir el escenario de la funcionalidad que se está probando, ¡de forma clara! Por ejemplo: "Ingresar al sitio web con edad menor a 18 años"
- Precondiciones
  - Conjunto de características que tienen que cumplirse en el contexto de la funcionalidad, para que pueda ser ejecutada.
  - Deben ser valores concretos, específicos. Por ej: "El usuario Juan está logueado con permiso de administrador." - "La fecha actual es 25/10/2022" - "Las coordenadas del GPS son: 41°24'12.2" N" - "La tarjeta tiene el número: XXXX XXXX XXXX XXXX"
  - Los datos que se seleccionan de entre un grupo determinado (por ejemplo forma de pago), son precondiciones que deben estar previamente cargadas en el sistema.
- Pasos
  - Siempre arrancan seleccionando la opción que desencadena la funcionalidad: "El *nombreUsuario* ingresa a la opción …"
  - Después: "El *nombreUsuario* ingresa/selecciona …
  - No tiene que haber ambigüedad, pensar que otra persona los tiene que leer y ejecutar.
  - Normalmente finalizan con un botón de confirmación o algo así.
- Resultado esperado
  - Puede mostrar varias cosas el sistema.
  - Deben ser resultados concretos, específicos.

- Suelen ser mensajes de advertencia/error, listados, posición en el mapa, etc. pero siempre indicando con un dato particular. Por ejemplo: El sistema muestra el mensaje de advertencia “Debe seleccionar una fecha”.
- Se debe relacionar la salida con el paso en la cual se produce.

Caja blanca

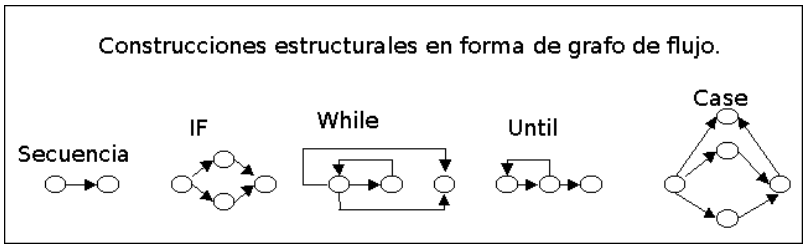
En estos se dispone del código (puede ser también pseudocódigo o un diagrama de flujo). Nos permiten diseñar casos de prueba, maximizando la cantidad de defectos con la menor cantidad de casos de prueba posibles.

Tiene dos fallas: La primera es que el número de caminos lógicos únicos puede ser sumamente grande, tomando muchísimo tiempo de probar absolutamente todas de ellas. La segunda falla es que las pruebas de camino exhaustivas no aseguran que el programa sea el correcto para las especificaciones, tampoco detecta caminos faltantes ni determina errores de datos sensibles.

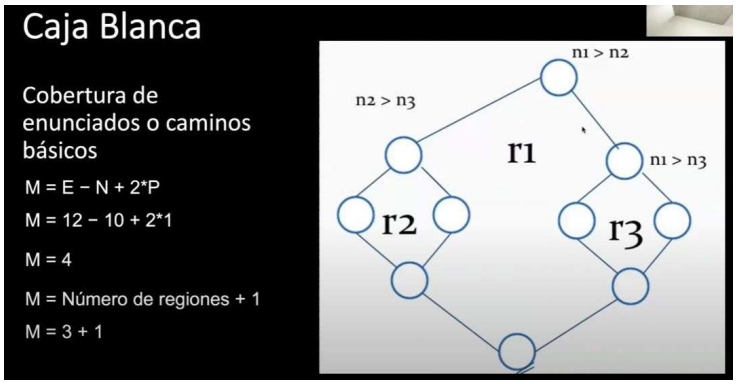
Hay distintas coberturas con nivel diferente (la forma de recorrer los distintos caminos que provee el código para desarrollar una funcionalidad):

Cobertura de enunciados o caminos básicos

- Objetivo
  Encontrar todos los caminos independientes de una funcionalidad que deben recorrerse (testear) al menos una vez, para probar cada uno de ellos con casos de prueba.
- Permite obtener una métrica denominada *complejidad ciclomática* (M), que representa la cantidad de caminos independientes que posee una funcionalidad, y nos da un límite inferior para el número de casos de prueba que hay que construir, para ejecutar todas las instrucciones de la funcionalidad al menos una vez.
- El procedimiento es:
  1. En primer lugar, se requiere representar la funcionalidad a través de un grafo de flujo (los algoritmos que implementen métodos recursivos quedan fuera de esta prueba de cobertura)



  2. Luego se calcula la complejidad ciclomática (M). Para esto hay dos formas:
     a. M = E - N + 2*P, siendo:
        - M = complejidad ciclomática.
        - E = número de aristas del grafo.
        - N = número de nodos del grafo.
        - P = número de componentes conexos o nodos de salida.
     b. También se puede obtener la complejidad ciclomática como M = número de regiones cerradas + 1.



En el ejemplo, la cantidad mínima de casos de prueba para garantizar la cobertura de enunciados es de 4. Esto no quita que se puedan construir más de 4, pero esa es la menor cantidad de casos de prueba, que me permite maximizar la detección de defectos.

3. Se define el conjunto mínimo de caminos independientes (asignando valores que provocan ir tomando cada uno de los caminos de la funcionalidad). No son casos de prueba. Siguiendo el ejemplo:

TC 1	TC 2	TC 3	TC 4
N1 = 8	N1 = 8	N1 = 4	N1 = 4
N2 = 4	N2 = 4	N2 = 8	N2 = 8
N3 = 4	N3 = 8	N3 = 4	N3 = 8

4. Luego se diseñan y ejecutan los casos de prueba que permitan la ejecución de cada uno de los caminos identificados, donde los datos de la tabla de arriba se mapean a las precondiciones o a valores que ingresa el usuario en los casos de prueba (dependiendo de cómo es la funcionalidad).

5. Se ejecutan los casos de prueba y se comprueba que los resultados sean los esperados.

- Dependiendo del valor de la complejidad ciclomática (M), el programa puede entrar en una de las siguientes clasificaciones (heurística):

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

Cobertura de sentencias

- Sentencia → cualquier instrucción como asignación de variable, invocación de métodos, etc. Las estructuras de control NO son sentencias, son decisiones.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan pasar, ejecutar o evaluar todas las sentencias.
- Ejemplo: Partiendo de la siguiente porción de código, podemos determinar que existen 2 sentencias (de asignación de variables). Entonces, para ejecutar o evaluar las sentencias, es necesario solo un caso de prueba, asignando valores a "A", "C", "B" y "D" es posible ejecutar ambas sentencias ya que las condiciones de IF son independientes entre sí.

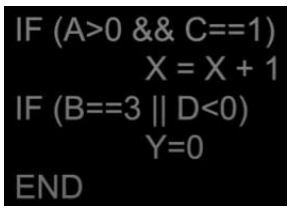
```
IF (A>0 && C==1)
        X = X + 1
IF (B==3 || D<0)
        Y=0
END
```

TC1
A=5
C=1
B=3
D=-3

Cobertura de decisión

- Decisión: estructura de control concreta, y dentro de cada decisión existen combinaciones que están unidas por operaciones booleanas de condiciones.
- Objetivo → buscar la cantidad mínima de casos de prueba que me permitan evaluar todas las ramas de las decisiones. Esto apunta a que el código tome cada rama de forma correcta, es decir, evaluar que la decisión derive en la rama del true y en la rama del false cuando corresponda en cada caso.
- En una decisión IF, si o si necesitamos dos casos de prueba: rama true y rama false.

- Las decisiones son las que se encuentran dentro de los paréntesis de los IF, y cada uno de los términos son condiciones, es decir A>0 y C==1, y el conjunto de las condiciones unidas por el booleano, representan una combinación (A>0 && C==1).
- En esta cobertura, no interesa qué condición (o combinación de condiciones) hay dentro de la decisión, siempre y cuando se garantice que la decisión es evaluada en su rama verdadera y en su rama falsa.
- En el ejemplo, con tan solo dos casos de prueba es suficiente, ya que las decisiones son independientes entre sí (no están anidados), por lo que independientemente de la rama que se evalúe en la primera decisión, voy a poder valuar cualquiera en la segunda. Entonces en el TC1 se evalúa la rama de "true" en ambas decisiones, y en TC2 se evalúa la rama de "false" en la dos.



TC1	TC2
A=5 C=1 B=3 D=-3	A=5 C=5 B=5 D=5

Cobertura de condición

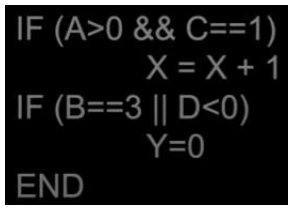
- Condiciones: son cada una de las evaluaciones lógicas dentro de una decisión, que están unidas por operadores lógicos.
- Objetivo → buscar la cantidad mínima de casos de pruebas que me permitan evaluar cada una de las condiciones, en su valor verdadero y en su valor falso, independientemente de que rama se tome en la decisión en la que se encuentre la condición (recordar que la decisión está formada por 1 o más condiciones combinadas de forma booleana).

TC1	TC2
A=0 C=1 B=3 D=-3	A=5 C=5 B=5 D=5

- Siguiendo el ejemplo anterior, son necesarios 2 casos de prueba, ya que las condiciones son independientes y se pueden evaluar todas las condiciones en true en una prueba, y todas las condiciones en false en otra prueba, debido a que no importa la rama que tome cada decisión.

Cobertura de decisión condición

- Objetivo → evaluar las ramas verdaderas y falsas de las decisiones, y a su vez evaluar valores verdaderos y falsos de las condiciones.
- Siguiendo el ejemplo anterior, se necesitan 2 test cases, los cuales permiten evaluar las condiciones y decisiones al mismo tiempo en un mismo caso de prueba, con los valores de la imagen.



TC1	TC2
A=5 C=1 B=3 D=-3	A=0 C=5 B=5 D=5

Cobertura múltiple

- Objetivo → evaluar el combinatorio de todas las condiciones, en todos sus valores de verdad posibles.
- Por ejemplo: Si tuviésemos un caso como el representado en el grafo de flujo, los casos de prueba van a ser 7, ya que solo el valor de A y B verdaderos al mismo tiempo, permitenejecutar escenarios en donde se incluye la decisión de C y D. Este es un escenario con decisiones dependientes entre sí, si fuesen independientes, con 4 pruebas alcanza.

1 DECISIÓN	A	F	F	V	V	V	V	V
	B	F	V	F	V	V	V	V
2 DECISIÓN	C	-	-	-	V	V	F	F
	D	-	-	-	V	F	V	F

Tipos de prueba

Smoke Test

Es un tipo de prueba que se hace para validar que no haya fallas groseras, de gran magnitud, en el producto de software. Se realiza antes de comenzar el ciclo 0. Nos ahorra empezar a testear formalmente si encuentra un error, porque todavía hay algo que corregir.

Consiste en una corrida rápida para ver si el producto está en condiciones de pasar al ciclo de prueba.

Testing Funcional

Controla que el software se comporte de la misma manera que lo especificado en la documentación, cumpliendo con las funcionalidades y características definidas. Se basa en los requerimientos funcionales y el proceso de negocio. Hay testing funcional basado en dos aspectos:

- Basado en requerimientos: cuando se prueban requerimientos específicos, apunta a probar una funcionalidad sola (utilizan a los requisitos definidos en una ERS o los acuerdos que contienen las pruebas de usuario y los criterios de aceptación de una US para realizar las pruebas.)
- Basado en proceso de negocio: cuando se prueba un proceso de negocio completo, es decir, se prueba todo el proceso. Por ejemplo, en una venta se prueba la búsqueda del artículo, la selección y facturación del mismo.

Testing No Funcional

Se basa en cómo trabaja el sistema haciendo foco en los requerimientos no funcionales (foco en el "como", no en el "que"). Son las pruebas más complejas, por su gran dependencia al entorno del sistema, por lo que debe ser lo más parecido posible al del cliente. Sin embargo, se tienen características que no pueden probarse, como el ancho de banda del internet, la seguridad o performance en una determinada situación. Y se pueden solucionar con las pruebas de aceptación. Incluye varios tipos de prueba como:

- Performance: Se ve el tiempo de respuesta (escenario esperado respecto a los tiempos de respuesta) y la concurrencia. Deben pasar esta prueba sí o sí.
- Carga: no solo mira performance, mira el comportamiento de los dispositivos de hardware (procesadores, discos, etc.) y de las comunicaciones.
- Stress: queremos forzar al sistema para que falle, se lo somete a condiciones más allá de las normales. Se ve el tiempo que hay que esperar para probar nuevamente el sistema y la robustez del sistema (tiempo de recuperación).
- Mantenimiento: para ver si el producto está en condiciones de evolucionar, se controla que haya documentación, manual de configuración, etc. Se observa la facilidad que existe para corregir un defecto.
- Usabilidad: que sea cómodo para el usuario.
- Portabilidad: se prueba en los distintos entornos acordados con el cliente.
- Fiabilidad: probamos que podemos depender del sistema. Resultados que se obtienen, seguridad física del software.
- De interfaz de usuario: suelen ser más complejas las GUI's que las interfaces de comandos.
- De configuración.

Test Driven Development – TDD

Este es un tipo de proceso en el que nos enfocamos en construir primero la prueba unitaria cuando tengo los requerimientos y luego codear el componente. Si pienso en las pruebas después tengo menos errores, el fundamento filosófico de esto es que si no tengo claro que tengo que hacer no puedo crear pruebas para eso.
Es una técnica de desarrollo del software que involucra dos prácticas:

1. **Test first development:**
   en esta técnica primero se escriben las pruebas unitarias referentes a la característica de producto a implementar. Definidas las pruebas unitarias, se piensa en la codificación necesaria para que las pruebas se

ejecuten con éxito. Dado que las pruebas unitarias prueban unidades concretas de código, esta técnica obliga al desarrollador a modularizar su codificación haciendo que los métodos o clases a probar tengan una única responsabilidad. Además de pruebas unitarias, pueden incluirse en primera instancia pruebas de integración.

2. **Refactoring:**
   esta técnica consiste en reestructurar el código existente sin modificar el comportamiento que el mismo provee. Implica la mejora de aspectos no funcionales del software, cuyo objetivo es mejorar la claridad del código y reducir su complejidad, con el fin de hacerlo más mantenible.